Speicherverwaltung, Segmente

- Codesegment
  - enthält Programmcode
  - keine Veränderung während der Ausführung (i.d.R.)
- Stacksegment
  - enthält lokale Daten in Abhängigkeit der dynamischen Schachtelung im Programmablauf
  - unproblematische, schnelle Reservierung und Freigabe durch LIFO-Prinzip
- Datensegment (für globale Daten)

  (a) statischer Bereich: fest definiert im Programmcode

  (b) dynamischer Bereich, Heap:
  - entwickelt sich ungeordnet im Programmablauf
  - abhängig von Art der Speicherverwaltung:

    (i) explizit: Reservierungen, Freigaben

    (ii) implizit: Definition eines Objekts, Garbage Collection

Explizite Speicherverwaltung

- Linux-Betriebssystemfunktion `brk`:
  - vergrößert das Datensegment des Prozesses
- C-Bibliotheksfunktionen für explizite Speicherverwaltung
  - `malloc(n)`
    - reserviert einen Bereich der Größe `n` aus dem Heap und gibt seine Anfangsadresse zurück
    - ruft `brk` auf, falls Datensegment zu klein
  - `free(p)`
    - gibt Speicher mit Anfangsadresse `p` frei

Blöcke und Chunks (1)

- Block
  - Speicherbereich für das Datenobjekt
  - gelesen und beschrieben vom Anwendungsprogramm
- Chunk
  - Header + Block
  - Header enthält Verwaltungsinformation
    - Größe des Chunks
    - Markierungsbit: Block ist frei (nicht reserviert)
  - Restriktion an Anfangsadresse
    - Alignment des Blocks (z.B. Adresse durch 8, 16 teilbar)
    - minus Größe des Headers

Blöcke und Chunks (2)

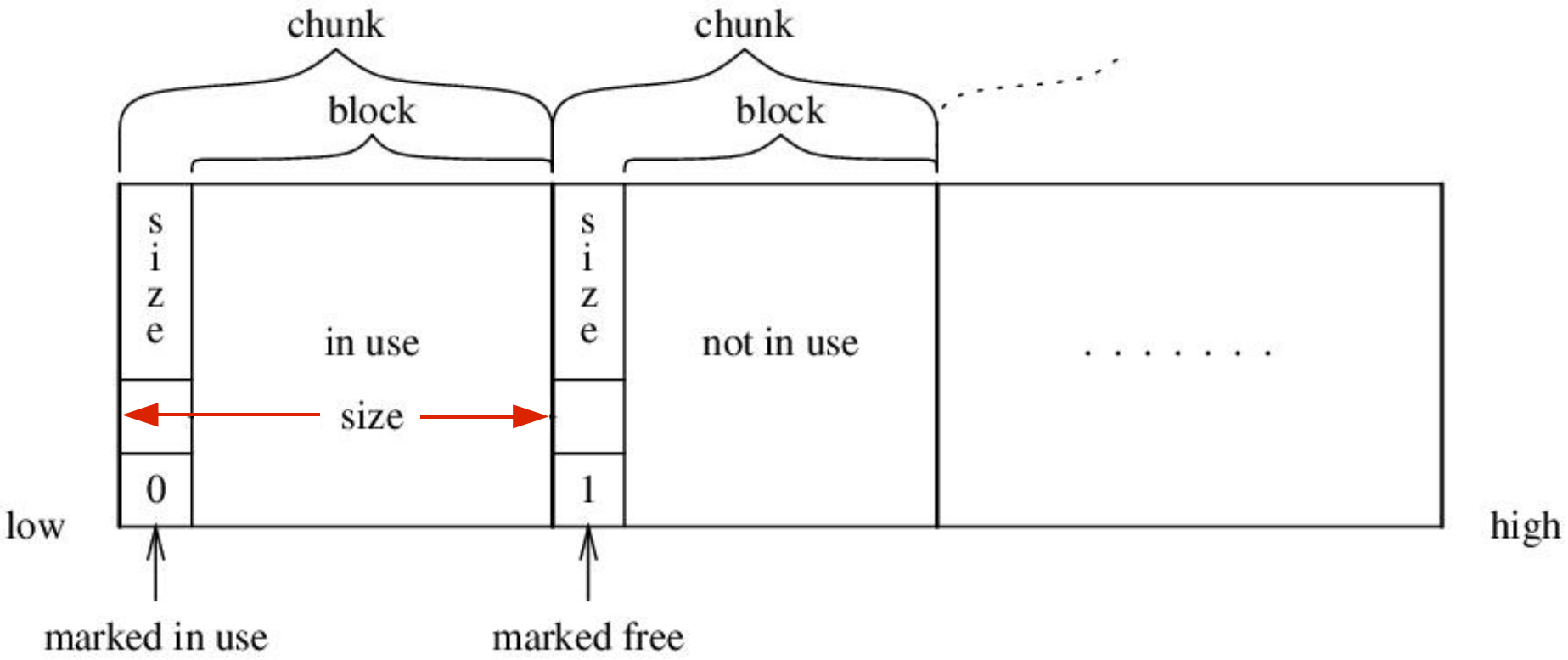

Figure 5.1 Memory structure used by the malloc/free mechanism.

Fehler im Buch: size muss Größe des Chunks sein

Speicherreservierung (1)

1. Springen von Chunk zu Chunk, bis freier Block ausreichender Größe gefunden (Existenz vorausgesetzt)
   - Block frei $\Leftrightarrow$ `.free=True` (Markierungsbit)
   - nächster Chunk: Adressdistanz im Größenfeld
2. Aufteilung des Chunks
   (a) einer für den zu reservierenden Block
   (b) einer für den nicht gebrauchten Restbereich
3. Setze Größenfelder und Belegungsmarkierungen der Teile
   (a) `.free := False`
   (b) `.free := True`

Speicherreservierung (2)

- falls kein freier Block ausreichender Größe gefunden wurde:
  - sukzessive Suche nach zwei benachbarten freien Chunks und Verschmelzung, bis ausreichend großer Chunk entstanden ist
  - ausreichend großer Chunk enstanden?
    - ja: Aufteilung, Header-Felder setzen und Rückgabe
    - nein: `Solution_to_out_of_memory_condition()`, z.B. Betriebssystemaufruf zur Vergrößerung des Datensegments (Unix: `brk()`)

Implementierung `Malloc/Free` (1)

- Initialisierung des ersten (zu Beginn einzigen) Chunks:

```
SET the polymorphic chunk pointer First_chunk pointer TO
    Beginning of available memory;
SET the polymorphic chunk pointer One past available memory TO
    Beginning of available memory + Size of available memory;

SET First_chunk pointer .size TO Size of available memory;
SET First_chunk pointer .free TO True;
```

Implementierung `Malloc/Free` (2)

- Reservierung eines neuen Blocks: `Malloc()`
  - Argument: gewünschte Größe
  - Rückgabe: Anfangsadresse des reservierten Speicherbereichs

Suche eines freien Blocks

```
FUNCTION Malloc (Block size) RETURNING a polymorphic block pointer:
    SET Pointer TO Pointer to free block of size (Block size);
    IF Pointer /= Null pointer: RETURN Pointer;

    Coalesce free chunks;
    SET Pointer TO Pointer to free block of size (Block size);
    IF Pointer /= Null pointer: RETURN Pointer;

    RETURN Solution to out of memory condition (Block size);
```

Verschmelzen freier benachbarter Blöcke

z.B. BS-Aufruf: Vergrößerung des Datensegments

Implementierung `Malloc/Free` (3)

- Freigabe eines nicht mehr gebrauchten Blocks: `Free()`
  - Argument: Anfangsadresse des freizugebenden Bereichs

```
PROCEDURE Free (Block pointer):
    SET Chunk pointer TO Block pointer - Administration size;
    SET Chunk pointer .free TO True;
```

Hilfsfunktionen (1)

- Suche eines freien Blocks, aufgerufen von `Malloc`:

```
FUNCTION Pointer to free block of size (Block size)
    RETURNING a polymorphic block pointer:
    // Note that this is not a pure function
    SET Chunk pointer TO First_chunk pointer;
    SET Requested chunk size TO Administration size + Block size;

    WHILE Chunk pointer /= One past available memory:
        IF Chunk pointer .free:
            IF Chunk pointer .size - Requested chunk size >= 0:
                // large enough chunk found:
                Split chunk (Chunk pointer, Requested chunk size);
                SET Chunk pointer .free TO False;
                RETURN Chunk pointer + Administration size;
        // try next chunk:
        SET Chunk pointer TO Chunk pointer + Chunk pointer .size;
    RETURN Null pointer;
```

Ende des Datensegments

Hilfsfunktionen (2)

- Chunk gefunden, der groß genug ist:
  ⇒ Aufteilung, um nicht gebrauchten Rest wieder freizugeben

```
PROCEDURE Split chunk (Chunk pointer, Requested chunk size):
    SET Left_over size TO Chunk pointer .size - Requested chunk size;
    IF Left_over size > Administration size:
        // there is a non-empty left-over chunk
        SET Chunk pointer .size TO Requested chunk size;
        SET Left_over chunk pointer TO
            Chunk pointer + Requested chunk size;
        SET Left_over chunk pointer .size TO Left_over size;
        SET Left_over chunk pointer .free TO True;
```

Hilfsfunktionen (3)

- Keinen Chunk gefunden, der groß genug:
  ⇒ Verschmelzen freier, benachbarter kleinerer Chunks

```
PROCEDURE Coalesce free chunks:
    SET Chunk pointer TO First_chunk pointer;

    WHILE Chunk pointer /= One past available memory:
        IF Chunk pointer .free:
            Coalesce with all following free chunks (Chunk pointer);
        SET Chunk pointer TO Chunk pointer + Chunk pointer .size;
```

Hilfsfunktionen (4)

- Verschmelzungsprozess wird fortgesetzt, solange benachbarte, freie Chunks existieren

```
PROCEDURE Coalesce with all following free chunks (Chunk pointer):
    SET Next_chunk pointer TO Chunk pointer + Chunk pointer .size;
    WHILE Next_chunk pointer /= One past available memory
        AND Next_chunk pointer .free:
        // Coalesce them:
        SET Chunk pointer .size TO
            Chunk pointer .size + Next_chunk pointer .size;
        SET Next_chunk pointer TO Chunk pointer + Chunk pointer .size;
```

Optimierungen (1)

- Suche des nächsten freien Chunks
  - freie Chunks werden in einer Liste verkettet
  - als Speicherplatz für Nachfolgezeiger dient freier Block
- Finden eines Chunks, der groß genug ist
  - Aufteilung der Größenbereiche, die vorkommen können
    - Zweierpotenzen in einem realistischen Bereich
    - im Programm verwendete Record-Größen
  - separate Liste für jeden Größenbereich

Optimierungen (2)

- Vergrößerung des Datensegments um ein Vielfaches einer Chunkgröße
  - Arrays für Chunks der gleichen Größe
  - Menge von Arrays für jeden Recordtyp im Programm
- Gemeinsame Speicherung von Blöcken gleicher Größe
  - kein Overhead für Blockgröße, nur Belegungsbit
  - zusätzliche Freilisten zur schnellen Allokation und Freigabe
  - Erinnerung: Verzeigerung der Liste benutzt Speicherplatz der freien Blöcke

Record-orientierte Blockreservierung (1)

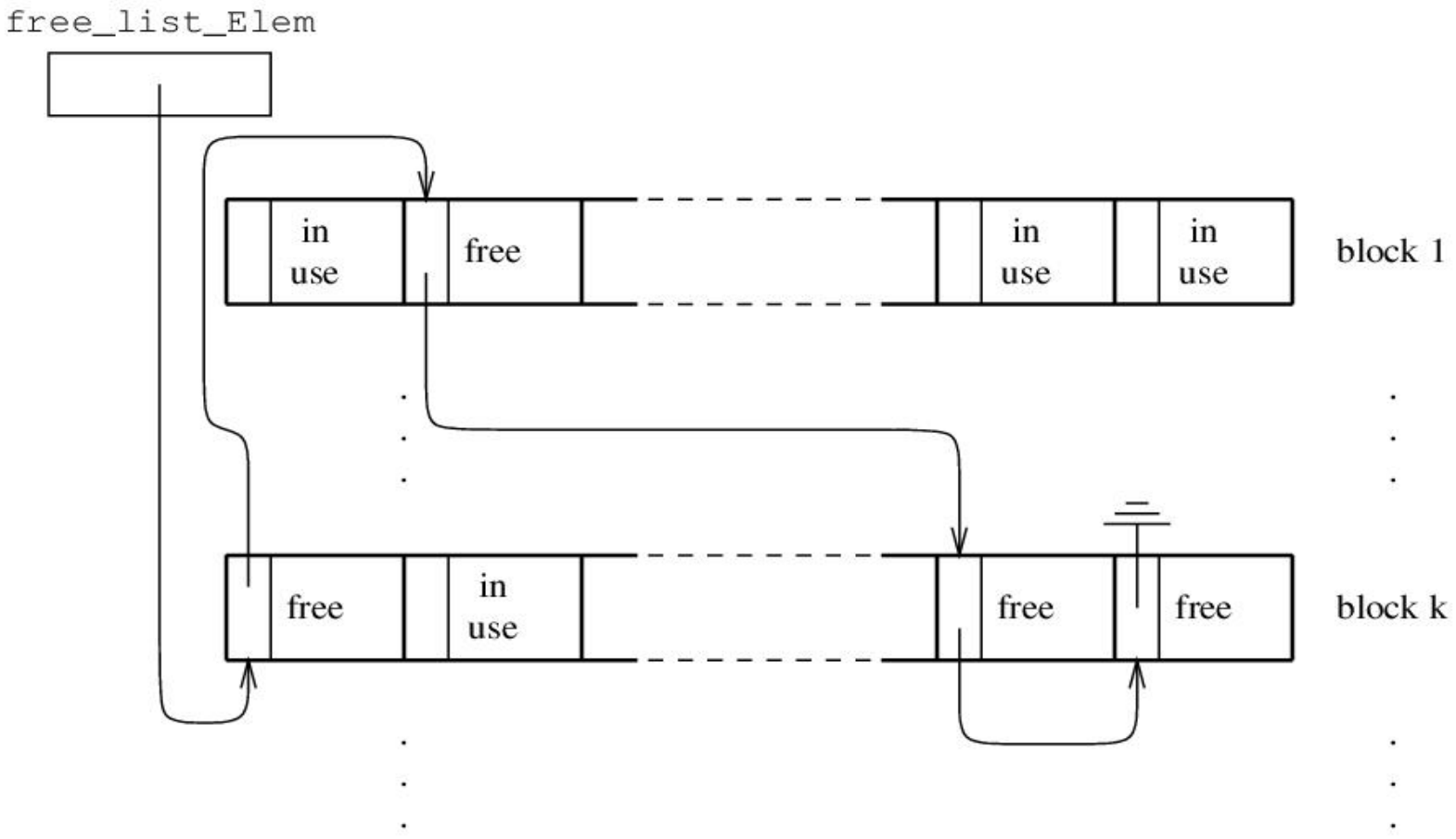

Figure 5.5 List of free records in allocated blocks.

Record-orientierte Blockreservierung (2)

- Reservierung eines Blocks für Recordtyp T:

```
FUNCTION New T () RETURNING a pointer to an T:
    IF Free list for T = No T:
        // Acquire a new block of records:
        SET New block [1 .. Block factor for T] TO
            Malloc (Size of T * Block factor for T);
        // Construct a free list in New block:
        SET Free list for T TO address of New block [1];
        FOR Record count IN [1 .. Block factor for T - 1]:
            SET New block [Record count] .link TO
                address of New block [Record count + 1];
        SET New block [Block factor for T] .link TO No T;

    // Extract a new record from the free list:
    SET New record TO Free list for T;
    SET Free list for T TO New record .link;

    // Zero the New record here, if required;
    RETURN New record;
```

Record-orientierte Blockreservierung (3)

- Freigabe eines Blocks für Recordtyp T:

```
PROCEDURE Free T (Old record):
    // Prepend Old record to free list:
    SET Old record .link TO Free list for T;
    SET Free list for T TO address of Old record;
```

Extensible Arrays (1)

- spezielles Array
  - “unbegrenzt” Elemente am Ende hinzufügen
  - Vorteil: Zugriff per Index in konstanter Zeit
  - Nachteil: löschen von Elementen aufwändig
- Verwendung
  - im Compiler
    - Hashtabellen
    - bearbeiten einer Sequenz unbekannter Länge
    - erzeugen von Objektcode
  - im Laufzeitsystem
  - bei Simulationen: Heap für Event-Queue

Extensible Arrays (2)

- beim Erweitern muss das alte Array umkopiert werden
- Parameter für Berechnung amortisierender Kosten
  - α: Kosten für Allokation eines neuen Arrays
  - γ: anteilige Kosten für das Kopieren eines Elements
- Erweiterung um jeweils μ Elemente: $C(n+\mu)=C(n)+\alpha+\gamma n$

  ⇒ quadratische Komplexität: $C(n)=\frac{\gamma}{2\mu}n^2+\left(\frac{\alpha}{\mu}-\frac{\gamma}{2}\right)n$
- Erweiterung um jeweils den Faktor β: $C(\beta n)=C(n)+\alpha+\gamma n$

  ⇒ nur lineare Komplexität: $C(n)=\frac{\gamma}{\beta-1}n+\frac{\alpha}{\ln(\beta)}\ln(n)$

Implizite Speicherverwaltung: Motivation

- Dangling Reference (verfrühte explizite Freigabe):
  - schwer zu lokalisieren
    - inkorrekter Heapzugriff u.U. viel später im Programmablauf
    - Auswirkung der fehlerhaften Daten u.U. noch viel später
    - Fehler schwer reproduzierbar, da vom Heapzustand abhängig
  - Sprachen: C, C++
- Feature für das Anwendungsprogramm
  - erzeugen neuer Objekte und kopieren von Referenzen auf Objekte
  - vergessen der Objekte, wenn sie nicht mehr gebraucht werden, keine explizite Freigabe
  - Sprachen: Java, deklarative Sprachen (Haskell, Prolog)

Implizite Speicherverwaltung, Methoden

- Garbage Set (nicht mehr benutzte Chunks):
  - Approximationen
    1. Chunks, auf die keine Zeiger verweisen
    2. Chunks, die nicht über Daten außerhalb des Heaps erreichbar sind
- Methoden

	reference counting	mark and scan	two-space copying
Approximation	1	2	2
Schwierigkeitsgrad	gering	hoch	mittel
Zeigerüberwachung	ja	nein	nein
Laufzeiteffizienz	akzeptabel	akzeptabel	sehr gut
Speichereffizienz	zirkuläre Strukturen nie entsorgt	optimal	doppelter Speicherverbrauch
Freiliste	aufgefüllt	aufgefüllt	neu angelegt

Implizite Speicherverwaltung, Begriffe

- Program Data Area: Programmdatenbereich außerhalb des Heaps
- Root Set: Menge der Zeiger im Programmdatenbereich
- Pointer Layout: Positionierung der Zeiger in einem Chunk
- Self-Descriptive Chunk: enthält Information über Pointer Layout
  - entweder wenn alle Chunks ein einheitliches Format haben
  - oder über eine Typbeschreibung im Chunk-Deskriptor
- Pointer Validity: Gültigkeit der Zeiger
  - nur wichtig an Stellen, an denen der Kollektor aktiv wird
  - wird über Typkopplung und Initialisierung erzwungen
- Speicherfragmentierung:
  - Problem: genug freier Speicher, aber nicht zusammenhängend
  - Lösung: Zusammenfassung freier Chunks nach Verschiebung(!) (Kompaktifizierung)

Scheduling-Varianten der Speicherbereinigung

- one-shot:
  - einfach, keine Interaktion mit dem Anwenderprogramm
  - Aktivierungszeiten nicht vorhersagbar
  - kann Programmausführung zeitweise blockieren
- on-the-fly / incremental:
  - komplizierter, aber weniger störend bei der Ausführung
  - vorbereitende Aktionen bei jedem Aufruf von Malloc und/oder Free, um entsprechende Chunks auffindbar zu machen
  - braucht im Extremfall one-shot-Kollektor zusätzlich
- nebenläufig
  - Speicherbereinigung in separatem Prozess
  - Algorithmus muss Interferenz mit Anwendungsprogramm berücksichtigen
  - besonders interessant für Multi-Core-Prozessoren

Dekomposition des Problems

Identifikation ...

1. der Zeiger des Root Set und deren Typen (Compiler)
2. der Zeiger in einem Chunk und deren Typen (Compiler)
3. aller erreichbaren Chunks über (1.) und (2.) (Laufzeitsystem)

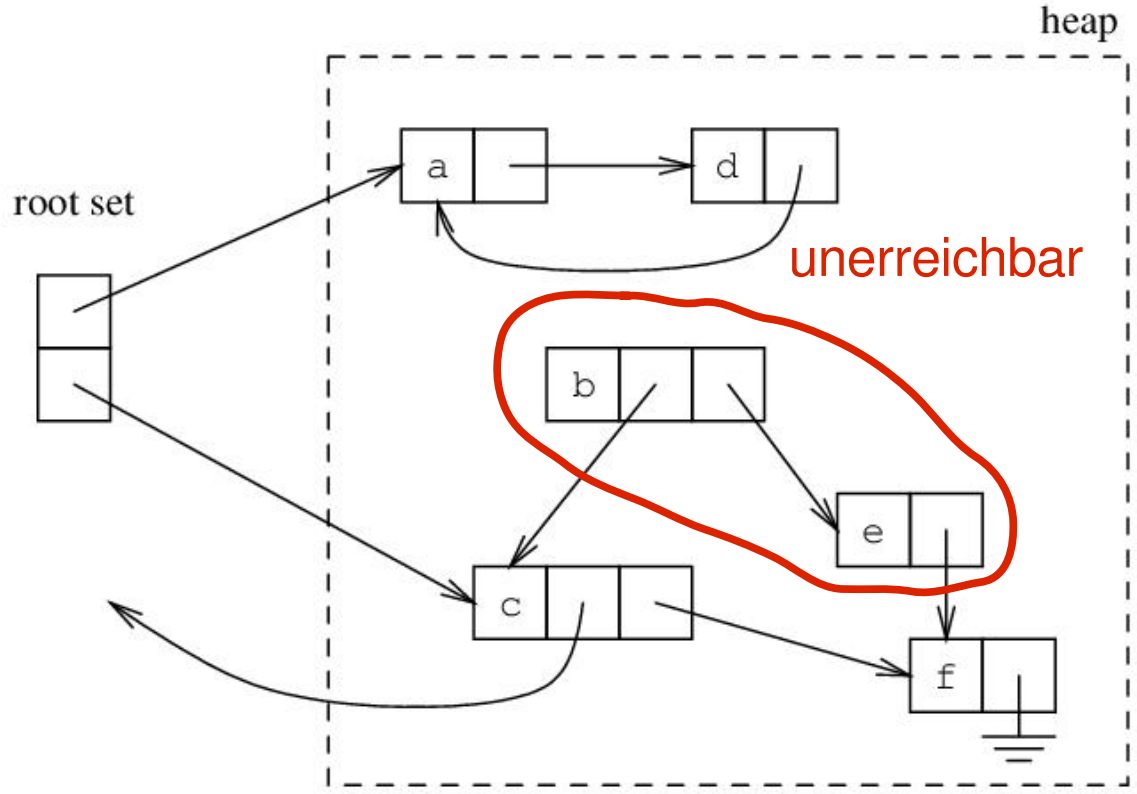

Figure 5.6 A root set and a heap with reachable and unreachable chunks.

Vereinfachende Annahmen

- der Kollektor arbeitet im Speicherbereich von Malloc und Free
- sowohl explizite als auch implizite Freigabe ist möglich:
  - explizit: auf Grund von speziellem Wissen des Laufzeitsystems
  - implizit: im Rahmen einer separaten Analyse
- Chunkanfang ist aus Zeiger errechenbar
- alle Chunks sind self-descriptive

Reale Kollektoren sind komplizierter

- Anwenderdaten sind komplizierter:
  - in Daten und Chunks besteht viel mehr Heterogenität
  - Größe und Typ von Daten und Chunks dynamisch veränderbar
  - Zeiger nicht immer initialisiert
- Optimierungen sind ein Komplikationsfaktor:
  - reale Kollektoren müssen aggressiv optimiert werden, aber ... die voran gegangene Codeoptimierung eliminiert Wissen, das der Kollektor brauchen könnte

⇒ es ist sehr schwer, einen Kollektor korrekt zu bekommen!